Constraint-Based Analysis

Mayur Naik
CIS 573 – Fall 2019
Motivation

Designing an efficient program analysis is challenging

Program Analysis = Specification + Implementation

“What”
No null pointer is dereferenced along any path in the program.

“How”
Many design choices:
- forward vs. backward traversal
- symbolic vs. explicit representation
- . . .
Motivation

Designing an efficient program analysis is challenging

Program Analysis = Specification + Implementation

Nontrivial!
Consider null pointer dereference analysis:
- No null pointer assignments ($v = \text{null}$): forward is best
- No pointer dereferences ($v->\text{next}$): backward is best

“How”
Many design choices:
- forward vs. backward traversal
- symbolic vs. explicit representation
- . . .
What Is Constraint-Based Analysis?

Designing an efficient program analysis is challenging.

Program Analysis = Specification + Implementation

"What" Defined by the user in the constraint language.

"How" Automated by the constraint solver.
Benefits of Constraint-Based Analysis

• Separates analysis specification from implementation
  – Analysis writer can focus on “what” rather than “how”

• Yields natural program specifications
  – Constraints are usually local, whose conjunctions capture global properties

• Enables sophisticated analysis implementations
  – Leverage powerful, off-the-shelf solvers
Consider a dataflow analysis such as live variables analysis. If one expresses it as a constraint-based analysis, one must still decide:

- The order in which statements should be processed.
- What the gen and kill sets for each kind of statement are.
- In what language to implement the chaotic iteration algorithm.
- Whether to take intersection or union at merge points.
Consider a dataflow analysis such as live variables analysis. If one expresses it as a constraint-based analysis, one must still decide:

- The order in which statements should be processed.
- What the gen and kill sets for each kind of statement are.
- In what language to implement the chaotic iteration algorithm.
- Whether to take intersection or union at merge points.

QUIZ: Specification & Implementation

✓

✓
Outline of this Lesson

A constraint language: Datalog

Two static analyses in Datalog:

- **Intra-procedural** analysis: computing *reaching definitions*
- **Inter-procedural** analysis: computing *points-to information*
A Constraint Language: Datalog

• A declarative logic programming language

• Not Turing-complete: subset of Prolog, or SQL with recursion
  => Efficient algorithms to evaluate Datalog programs

• Originated as query language for deductive databases

• Later applied in many other domains: software analysis, data mining, networking, security, knowledge representation, cloud-computing, ...

• Many implementations: Logicblox, bddbddb, IRIS, Paddle, ...
Syntax of Datalog: Example

**Input Relations:**
edge(n:N, m:N)

**Output Relations:**
path(n:N, m:N)

**Rules:**
path(x, x).
path(x, z) :- path(x, y), edge(y, z).
A relation is similar to a table in a database. A tuple in a relation is similar to a row in a table.

**Input Relations:**
edge(n:N, m:N)

**Output Relations:**
path(n:N, m:N)

**Rules:**
path(x, x).
path(x, z) :- path(x, y), edge(y, z).
Syntax of Datalog: Example

Input Relations:
edge(n:N, m:N)

Output Relations:
path(n:N, m:N)

Rules:
path(x, x).
path(x, z) :- path(x, y), edge(y, z).
Deductive rules that hold universally (i.e., variables like x, y, z can be replaced by any constant). Specify “if ... then ...” logic.

Syntax of Datalog: Example

**Input Relations:**
edge(n:N, m:N)

**Output Relations:**
path(n:N, m:N)

**Rules:**
path(x, x).
path(x, z) :- path(x, y), edge(y, z).
Syntax of Datalog: Example

**Input Relations:**
edge(n:N, m:N)

**Output Relations:**
path(n:N, m:N)

**Rules:**
path(x, x).
path(x, z) :- path(x, y), edge(y, z).

(If TRUE,) there is a path from each node to itself.

If there is path from node x to y, and there is an edge from y to z, then there is path from x to z.
Semantics of Datalog: Example

Input Relations:
\( \text{edge}(n:N, m:N) \)

Output Relations:
\( \text{path}(n:N, m:N) \)

Rules:
\( \text{path}(x, x). \)
\( \text{path}(x, z) :- \text{path}(x, y), \text{edge}(y, z). \)

\[
\text{path} := \{ (x, x) \mid x \in N \}
\]
do
\[
\text{path} := \text{path} \cup \{ (x, z) \mid \exists y \in N: (x, y) \in \text{path} \text{ and } (y, z) \in \text{edge} \}
\]until \( \text{path} \) relation stops changing
Semantics of Datalog: Example

**Input Relations:**
edge(n:N, m:N)

**Output Relations:**
path(n:N, m:N)

**Rules:**
path(x, x).
path(x, z) :- path(x, y), edge(y, z).

**Input Tuples:**
edge(0, 1), edge(0, 2), edge(2, 3), edge(2, 4)

**Output Tuples:**
path(0, 0), path(1, 1), path(2, 2), path(3, 3), path(4, 4), path(0, 1), path(0, 2), path(2, 3), path(2, 4), path(0, 3), path(0, 4)
Semantics of Datalog: Example

**Input Relations:**
edge(n:N, m:N)

**Output Relations:**
path(n:N, m:N)

**Rules:**
path(x, x).
path(x, z) :- path(x, y), edge(y, z).

**Input Tuples:**
edge(0, 1), edge(0, 2), edge(2, 3), edge(2, 4)

**Output Tuples:**
path(0, 0), path(1, 1), path(2, 2), path(3, 3), path(4, 4), path(0, 1), path(0, 2), path(2, 3), path(2, 4), path(0, 3), path(0, 4)
Semantics of Datalog: Example

**Input Relations:**
edge(n:N, m:N)

**Output Relations:**
path(n:N, m:N)

**Rules:**
path(x, x).
path(x, z) :- path(x, y), edge(y, z).

**Input Tuples:**
edge(0, 1), edge(0, 2), edge(2, 3), edge(2, 4)

**Output Tuples:**
path(0, 0), path(1, 1), path(2, 2), path(3, 3), path(4, 4), path(0, 1), path(0, 2), path(2, 3), path(2, 4), path(0, 3), path(0, 4)
Semantics of Datalog: Example

**Input Relations:**
edge(n:N, m:N)

**Output Relations:**
path(n:N, m:N)

**Rules:**
path(x, x).

path(x, z) :- path(x, y), edge(y, z).

**Input Tuples:**
edge(0, 1), edge(0, 2), edge(2, 3), edge(2, 4)

**Output Tuples:**
path(0, 0), path(1, 1), path(2, 2), path(3, 3), path(4, 4), path(0, 1), path(0, 2), path(2, 3), path(2, 4), path(0, 3), path(0, 4)
QUIZ: Computation Using Datalog

Check each of the below Datalog programs that computes in relation \texttt{scc} exactly those pairs of nodes \((n1, n2)\) such that \(n2\) is reachable from \(n1\) AND \(n1\) is reachable from \(n2\).

- \(\texttt{scc}(n1, n2) :- \text{edge}(n1, n2), \text{edge}(n2, n1).\)
- \(\texttt{scc}(n1, n2) :- \text{path}(n1, n2), \text{path}(n2, n1).\)
- \(\texttt{scc}(n1, n2) :- \text{path}(n1, n3), \text{path}(n3, n2), \text{path}(n2, n4), \text{path}(n4, n1).\)
- \(\texttt{scc}(n1, n2) :- \text{path}(n1, n3), \text{path}(n2, n3).\)
QUIZ: Computation Using Datalog

Check each of the below Datalog programs that computes in relation \texttt{scc} exactly those pairs of nodes \((n1, n2)\) such that \(n2\) is reachable from \(n1\) AND \(n1\) is reachable from \(n2\).

- \(\texttt{scc}(n1, n2) :- \text{edge}(n1, n2), \text{edge}(n2, n1).\) 
- \(\checkmark \quad \texttt{scc}(n1, n2) :- \text{path}(n1, n2), \text{path}(n2, n1).\) 
- \(\checkmark \quad \texttt{scc}(n1, n2) :- \text{path}(n1, n3), \text{path}(n3, n2), \text{path}(n2, n4), \text{path}(n4, n1).\) 
- \(\texttt{scc}(n1, n2) :- \text{path}(n1, n3), \text{path}(n2, n3).\)
Outline of this Lesson

A constraint language: Datalog

Two static analyses in Datalog:

- Intra-procedural analysis: computing reaching definitions
- Inter-procedural analysis: computing points-to information
Dataflow Analysis in Datalog

• Recall the specification of \textit{reaching definitions analysis}:

\[
\begin{align*}
\text{OUT}[n] &= (\text{IN}[n] - \text{KILL}[n]) \cup \text{GEN}[n] \\
\text{IN}[n] &= \bigcup_{n' \in \text{predecessors}(n)} \text{OUT}[n']
\end{align*}
\]
Reaching Definitions Analysis in Datalog

**Input Relations:**
\[
\text{kill}(n:N, d:D)
\]

**Output Relations:**
\[
\begin{align*}
\text{Definition } d & \text{ is killed by statement } n. \\
\text{OUT}[n] & = (\text{IN}[n] - \text{KILL}[n]) \cup \text{GEN}[n] \\
\text{IN}[n] & = \bigcup_{n' \in \text{predecessors}(n)} \text{OUT}[n']
\end{align*}
\]
Reaching Definitions Analysis in Datalog

**Input Relations:**
- \( \text{kill}(n:N, d:D) \)
- \( \text{gen}(n:N, d:D) \)

**Output Relations:**

**Definition d is generated by statement n.**

\[
\text{OUT}[n] = (\text{IN}[n] - \text{KILL}[n]) \cup \text{GEN}[n]
\]

**Rules:**

\[
\text{IN}[n] = \bigcup_{n' \in \text{predecessors}(n)} \text{OUT}[n']
\]
Reaching Definitions Analysis in Datalog

Input Relations:
- `kill(n:N, d:D)`
- `gen(n:N, d:D)`
- `next(n:N, m:N)`

Output Relations:

Statement \( m \) is an immediate successor of statement \( n \).

\[
\text{OUT}[n] = (\text{IN}[n] - \text{KILL}[n]) \cup \text{GEN}[n]
\]

\[
\text{IN}[n] = \bigcup \text{OUT}[n']
\]

\( n' \in \text{predecessors}(n) \)
Reaching Definitions Analysis in Datalog

**Input Relations:**
- kill(n:N, d:D)
- gen (n:N, d:D)
- next(n:N, m:N)

**Output Relations:**

\[
\text{OUT}[n] = (\text{IN}[n] - \text{KILL}[n]) \cup \text{GEN}[n]
\]

**Rules:**

\[
\text{IN}[n] = \bigcup_{n' \in \text{predecessors}(n)} \text{OUT}[n']
\]
Reaching Definitions Analysis in Datalog

Input Relations:
- kill(n:N, d:D)
- gen (n:N, d:D)
- next(n:N, m:N)

Output Relations:
- in (n:N, d:D)

Rules:

Definition d may reach the program point just before statement n.

\[
\text{OUT}[n] = (\text{IN}[n] - \text{KILL}[n]) \cup \text{GEN}[n]
\]

\[
\text{IN}[n] = \bigcup_{n' \in \text{predecessors}(n)} \text{OUT}[n']
\]
Reaching Definitions Analysis in Datalog

Input Relations:
kill(n:N, d:D)
gen (n:N, d:D)
next(n:N, m:N)

Output Relations:
in (n:N, d:D)
out(n:N, d:D)

Rules:

\[
\text{OUT}[n] = (\text{IN}[n] - \text{KILL}[n]) \cup \text{GEN}[n]
\]

\[
\text{IN}[n] = \bigcup_{n' \in \text{predecessors}(n)} \text{OUT}[n']
\]

Definition \( d \) may reach the program point just after statement \( n \).
Reaching Definitions Analysis in Datalog

Input Relations:
kill(n:N, d:D)
gen(n:N, d:D)
next(n:N, m:N)

Output Relations:
in(n:N, d:D)
out(n:N, d:D)

Rules:
out(n, d) :- gen(n, d).
out(n, d) :- in(n, d), !kill(n, d).

\[
\text{OUT}[n] = (\text{IN}[n] - \text{KILL}[n]) \cup \text{GEN}[n]
\]

\[
\text{IN}[n] = \bigcup_{n' \in \text{predecessors}(n)} \text{OUT}[n']
\]
Reaching Definitions Analysis in Datalog

Input Relations:
- kill(n:N, d:D)
- gen(n:N, d:D)
- next(n:N, m:N)

Output Relations:
- in(n:N, d:D)
- out(n:N, d:D)

OUT[n] = (IN[n] - KILL[n]) ∪ GEN[n]

IN[n] = \bigcup_{n' \in \text{predecessors}(n)} \text{OUT}[n']

Rules:
- out(n, d) :- gen(n, d).
- out(n, d) :- in(n, d), !kill(n, d).
- in(m, d) :- out(n, d), next(n, m).
**Reaching Definitions Analysis: Example**

**Input Relations:**
- kill(n:N, d:D)
- gen(n:N, d:D)
- next(n:N, m:N)

**Output Relations:**
- in(n:N, d:D)
- out(n:N, d:D)

**Rules:**
- out(n, d) :- gen(n, d).
- out(n, d) :- in(n, d), !kill(n, d).
- in(m, d) :- out(n, d), next(n, m).

![Diagram]

1: entry
2: x = 8
3: (x != 1)?
4: x=x-1
5: exit
Input Relations:
- kill(n:N, d:D)
- gen(n:N, d:D)
- next(n:N, m:N)

Output Relations:
- in(n:N, d:D)
- out(n:N, d:D)

Rules:
- out(n, d) :- gen(n, d).
- out(n, d) :- in(n, d), !kill(n, d).
- in(m, d) :- out(n, d), next(n, m).

Input Tuples:
- kill(4, 2),
- gen(2, 2), gen(4, 4),
- next(1, 2), next(2, 3),
- next(3, 4), next(3, 5),
- next(4, 3)

Reaching Definitions Analysis: Example

Diagram:
- Entry
- x = 8
- (x != 1)?
  - true
  - x = x - 1
  - false
- Exit
Reaching Definitions Analysis: Example

**Input Relations:**
- `kill(n:N, d:D)`
- `gen(n:N, d:D)`
- `next(n:N, m:N)`

**Output Relations:**
- `in(n:N, d:D)`
- `out(n:N, d:D)`

**Rules:**
- `out(n, d) :- gen(n, d).`
- `out(n, d) :- in(n, d), !kill(n, d).`
- `in(m, d) :- out(n, d), next(n, m).`

**Input Tuples:**
- `kill(4, 2),`
- `gen(2, 2), gen(4, 4),`
- `next(1, 2), next(2, 3),`
- `next(3, 4), next(3, 5),`
- `next(4, 3)`

**Output Tuples:**
- `in(3, 2), in(3, 4), in(4, 2),`
- `in(4, 4), in(5, 2), in(5, 4),`
- `out(2, 2), out(3, 2), out(3, 4),`
- `out(4, 2), out(4, 4), out(5, 2),`
- `out(5, 4)`

**Flow Chart:**
- **Entry:**
  - `x = 8`
  - `(x != 1)?`
  - **True:**
    - `x = x - 1`
  - **False:**
    - **Exit**
QUIZ: Live Variables Analysis

Complete the Datalog program below by filling in the rules for live variables analysis.

Input Relations:
kill(n:N, v:V)
gen (n:N, v:V)
ext(n:N, m:N)

Output Relations:
in (n:N, v:V)
out(n:N, v:V)

Rules:

: - , ! .
:- , ! .
:- , .
Complete the Datalog program below by filling in the rules for live variables analysis.

**Input Relations:**
- `kill(n:N, v:V)`
- `gen(n:N, v:V)`
- `next(n:N, m:N)`

**Output Relations:**
- `in(n:N, v:V)`
- `out(n:N, v:V)`

**Rules:**

\[
\begin{align*}
\text{in}(n, v) & \langle :- \quad \text{gen}(n, v) \quad . \\
\text{in}(n, v) & \langle :- \quad \text{out}(n, v) \quad , \quad ! \quad \text{kill}(n, v) \quad . \\
\text{out}(n, v) & \langle :- \quad \text{in}(m, v) \quad , \quad \text{next}(n, m) \quad .
\end{align*}
\]
Outline of this Lesson

A constraint language: Datalog

Two static analyses in Datalog:

• Intra-procedural analysis: computing reaching definitions

• Inter-procedural analysis: computing points-to information
Consider a flow-insensitive **may-alias analysis** for a simple language:

(function body) \( f(v) \) \{ s_1, \ldots, s_n \} 

(statement) \( s ::= v = \text{new} \ h \mid v = u \mid \text{return} \ u \mid v = f(u) \)

(pointer variable) \( u, v \)

(allocation site) \( h \)

(function name) \( f \)
Consider a flow-insensitive **may-alias analysis** for a simple language:

(function body) \( f(v) \) { \( s_1, \ldots, s_n \) }

(statement) \( s ::= v = \text{new } h \mid v = u \mid \text{return } u \mid v = f(u) \)

(pointer variable) \( u, v \)

(allocation site) \( h \)

(function name) \( f \)
Pointer Analysis in Datalog: Intra-procedural

Recall the specification:

Before:

\[ v = \text{new} \ h \]

After:

\[ v = u \]
Input Relations:
new (v:V, h:H)
assign(v:V, u:V)

Output Relations:

Rules:
**Pointer Analysis in Datalog: Intra-procedural**

**Before:**
- \( v = \text{new } h \)

**After:**
- \( v \rightarrow h \)
- \( u \rightarrow h2 \)

**Input Relations:**
- new \((v:V, h:H)\)
- assign\((v:V, u:V)\)

**Output Relations:**
- points\((v:V, h:H)\)

**Rules:**
Pointer Analysis in Datalog: Intra-procedural

Input Relations:
new  (v:V, h:H)
assign(v:V, u:V)

Output Relations:
points(v:V, h:H)

Rules:
points(v, h) :- new(v, h).

Before:
v = new h

After:
v = u

Diagram:

- Input Relations:
  - new(v:V, h:H)
  - assign(v:V, u:V)
- Output Relations:
  - points(v:V, h:H)
- Rules:
  - points(v, h) :- new(v, h).
Pointer Analysis in Datalog: Intra-procedural

Input Relations:
new (v:V, h:H)
assign(v:V, u:V)

Output Relations:
points(v:V, h:H)

Rules:
points(v, h) :- new(v, h).
points(v, h) :- assign(v, u),
               points(u, h).

Before:

After:
Consider a flow-insensitive may-alias analysis for a simple language:

(function body) \( f(v) \) \{ s1, ..., sn \}

(statement) \( s ::= v = \text{new } h \mid v = u \mid \text{return } u \mid v = f(u) \)

(pointer variable) \( u, v \)

(allocation site) \( h \)

(function name) \( f \)
Pointer Analysis in Datalog: Inter-procedural

```java
x = new h1;
y = f(x);
f(v) {
    u = v;
    return u;
}
```
Parameter passing and return can be treated as assignments!

**Input Relations:**
new (v:V, h:H)
assign(v:V, u:V)

**Output Relations:**
points(v:V, h:H)

**Rules:**
points(v, h) :- new(v, h).
points(v, h) :- assign(v, u), points(u, h).
**Pointer Analysis in Datalog: Inter-procedural**

**Input Relations:**
- `new(v:V, h:H)`
- `assign(v:V, u:V)`

**Output Relations:**
- `points(v:V, h:H)`

**Rules:**
- `points(v, h) :- new(v, h).`
- `points(v, h) :- assign(v, u), points(u, h).`

```plaintext
x = new h1;
y = f(x);
f(v) {
    u = v;
    return u;
}
```
### Pointer Analysis in Datalog: Inter-procedural

#### Input Relations:
- `new(v:V, h:H)`
- `arg(f:F, v:V)`
- `ret(f:F, u:V)`
- `assign(v:V, u:V)`
- `call(y:V, f:F, x:V)`

#### Output Relations:
- `points(v:V, h:H)`

#### Rules:
- `points(v, h) :- new(v, h).`  
- `points(v, h) :- assign(v, u), points(u, h).`
Input Relations:
assign(v:V, u:V) call(y:V, f:F, x:V)

Output Relations:
points(v:V, h:H)

Rules:
points(v, h) :- new(v, h).
points(v, h) :- assign(v, u), points(u, h).
### Pointer Analysis in Datalog: Inter-procedural

**Input Relations:**
- `new(v:V, h:H)`
- `arg(f:F, v:V)`
- `ret(f:F, u:V)`
- `assign(v:V, u:V)`
- `call(y:V, f:F, x:V)`

**Output Relations:**
- `points(v:V, h:H)`

**Rules:**
- `points(v, h) :- new(v, h).`
- `points(v, h) :- assign(v, u), points(u, h).`
- `points(v, h) :- call(_, f, x), arg(f, v), points(x, h).`

```
x = new h1;
y = f(x);
f(v) {
  u = v;
  return u;
}
```
Pointer Analysis in Datalog: Inter-procedural

Input Relations:

Output Relations:
points(v:V, h:H)

Rules:
points(v, h) :- new(v, h).
points(v, h) :- assign(v, u), points(u, h).
points(v, h) :- call(_, f, x), arg(f, v),
                points(x, h).
points(y, h) :- call(y, f, _), ret(f, u),
                points(u, h).
Check each of the below Datalog programs that computes in relation mustNotAlias each pair of variables (u, v) such that u and v do not alias in any run of the program.

- **mustNotAlias** (u, v) :- points(u, h1), points(v, h2), h1 != h2.
- mayAlias(u, v) :- points(u, h), points(v, h).
  **mustNotAlias** (u, v) :- !mayAlias(u, v).
- mayAlias(u, v) :- points(u, _), points(v, _).
  **mustNotAlias** (u, v) :- !mayAlias(u, v).
- common(u, v, h) :- points(u, h), points(v, h).
  mayAlias(u, v) :- common(u, v, _).
  **mustNotAlias** (u, v) :- !mayAlias(u, v).
QUIZ: Querying Pointer Analysis in Datalog

Check each of the below Datalog programs that computes in relation **mustNotAlias** each pair of variables \((u, v)\) such that \(u\) and \(v\) do not alias in any run of the program.

- mustNotAlias(u, v) :- points(u, h1), points(v, h2), h1 != h2.
- mayAlias(u, v) :- points(u, h), points(v, h).
- mustNotAlias(u, v) :- !mayAlias(u, v).

- mustNotAlias(u, v) :- !mayAlias(u, v).
- common(u, v, h) :- points(u, h), points(v, h).
- mayAlias(u, v) :- common(u, v, _).
- mustNotAlias(u, v) :- !mayAlias(u, v).
Context Sensitivity

\[
\begin{align*}
x &= \text{new } h1; \\
z &= \text{new } h2; \\
y &= f(x); \\
w &= f(z); \\
f(v) \{ \\
    &\quad u = v; \\
    &\quad \text{return } u; \\
}\end{align*}
\]

**Input Relations:**

assign(v:V, u:V) call(y:V, f:F, x:V)

**Output Relations:**

points(v:V, h:H)

**Rules:**

points(v, h) :- new(v, h).
points(v, h) :- assign(v, u), points(u, h).
points(v, h) :- call(_, f, x), arg(f, v), points(x, h).
points(y, h) :- call(y, f, _), ret(f, u), points(u, h).
**Context Sensitivity**

**Input Relations:**

**Output Relations:**
points(v:V, h:H)

**Rules:**
points(v, h) :- new(v, h).
points(v, h) :- assign(v, u), points(u, h).
points(v, h) :- call(_, f, x), arg(f, v), points(x, h).
points(y, h) :- call(y, f, _), ret(f, u), points(u, h).
x = new h1;
z = new h2;
y = f(x);
w = f(z);
f(v) {
    u = v;
    return u;
}

v = x
u = v
y = u

v = z
u = v
w = u
Context Sensitivity

\[
x = \text{new } h1;\\
z = \text{new } h2;\\
y = f(x);\\
w = f(z);\\
f(v) \{\\
    u = v;\\
    \text{return } u;\\
\}
\]

\[
v = x\\
u = v\\
y = u\\
\]

\[
v = z\\
u = v\\
w = u\\
\]

Imprecision!
Cloning-Based Inter-procedural Analysis

Achieves context sensitivity by **inlining** procedure calls

Cloning depth ↑: precision ↑ vs. scalability ↓
What about Recursion?

```
x = new h1;
z = new h2;
y = f(x);
w = f(z);

f(v) {
    if (*)
        v = f(v);
    return v;
}
```

Need **infinite** cloning depth to differentiate the points-to sets of x, y and w, z!
Summary-Based Inter-procedural Analysis

- Use the incoming program states to differentiate calls to the same procedure
  - Same incoming program states yield same outgoing program states for a given procedure
- As precise as cloning-based analysis with infinite cloning depth
# Other Constraint Languages

<table>
<thead>
<tr>
<th>Constraint Language</th>
<th>Problem Expressed</th>
<th>Example Solvers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datalog</td>
<td>Least solution of deductive inference rules</td>
<td>LogixBlox, bddbbdddb</td>
</tr>
<tr>
<td>SAT</td>
<td>Boolean satisfiability problem</td>
<td>MiniSat, Glucose</td>
</tr>
<tr>
<td>MaxSAT</td>
<td>Boolean satisfiability problem extended with optimization</td>
<td>open-wbo, SAT4j</td>
</tr>
<tr>
<td>SMT</td>
<td>Satisfiability modulo theories problem</td>
<td>Z3, Yices</td>
</tr>
<tr>
<td>MaxSMT</td>
<td>Satisfiability modulo theories problem extended with optimization</td>
<td>Z3</td>
</tr>
</tbody>
</table>
What Have We Learned?

- **Constraint-based analysis** and its benefits
- The **Datalog** constraint language
- How to express static analyses in Datalog
  - Analysis *logic* == *constraints* in Datalog
  - Analysis *inputs* and *outputs* == *relations* of tuples
- **Context-insensitive** and **context-sensitive** inter-procedural analysis